2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

ChameleonEC: Exploiting Tunability of Erasure Coding for Low-Interference Repair

Yuhui Cai†, Shiyao Lin†, Zhirong Shen†✉, Jiahui Yang† and Jiwu Shu†‡
†Xiamen University, ‡Minjiang University
✉Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn)

***Abstract*—Erasure coding provides fault tolerance in a storage-efficient manner, yet it introduces a high repair penalty. We uncover via trace-driven experiments that the substantial repair traffic in erasure coding is prone to entangling with the foreground traffic, thereby slowing down repair progress and downgrading service quality. We present ChameleonEC, a general mechanism that can assist a variety of erasure codes in realizing low-interference repair. ChameleonEC comprises the following design techniques: (i) repair task assignment, which decomposes a repair plan into multiple repair tasks and makes them co-exist harmoniously with the foreground traffic, so as to saturate unoccupied bandwidth and avoid bandwidth contentions; (ii) repair path establishment, which orchestrates elastic transmission routings over the dispatched repair tasks to instruct the repair; and (iii) straggler-aware re-scheduling, which timely re-tunes task transmissions and repair plans to bypass unexpected stragglers emerging in repair. We conduct extensive experiments on Amazon EC2, showing that ChameleonEC can accelerate the repair by 4.9–498.2% for various erasure codes under different real-world traces. ChameleonEC can also speed up the repair process by 25.4–73.5% under the storage-bottlenecked scenarios.**
[1]

I. Introduction

To protect data against unexpected failures, production systems [6], [8], [13], [19], [51], [53] popularly employ *erasure coding* [55], which can attain the same fault tolerance degree as *replication* with much less storage consumption [78]. At a high level, erasure coding encodes a number of fixed-size *data chunks* (often tens of MBs) to generate a few redundant chunks (called *parity chunks*), such that the original data chunks are always repairable, even some of the data and parity chunks are lost (Section II-A). Till now, erasure coding is popularly used to tolerate failures in persistent storage [19], [35], [79] and remote memory [42], [83], and shorten access latency in cluster caches [59], [75] and SSDs [57], [80].

While being storage-efficient, erasure coding amplifies the *repair traffic* (i.e., the data transmitted over networks for repair), as it needs to retrieve *multiple* surviving chunks to repair a *single* failed chunk. As failures may be correlated [22], [30], the repair process — once extended by the amplified repair traffic — will make the system vulnerable in the presence of subsequent failures. Therefore, to boost data repair, tremendous efforts have been made, which can be roughly classified into the following branches: (i) the construction of *repair-efficient erasure codes* [25], [29], [31], [35], [40], [54], [58], [62], [64] that are theoretically proved to reduce the repair traffic in coding theory and (ii) the design of *repair algorithms* [20], [32], [36], [45], [46], [48], [50], [67], [68] that exploit transmission parallelism to balance the bandwidth utilization across the whole systems.

Our observation is that most existing studies solely focus on the repair acceleration, yet limited attention is paid to the potential interference once the repair encounters the foreground traffic. Hence, when directly deploying them in storage systems, the introduced interference will delay the repair and degrade the access quality, which has also been discovered in Pangu of AliCloud [76] and Facebook Warehouse Cluster [61]. We further investigate the impact of the interference by evaluating the repair performance when the system is also serving requests from YCSB [24] clients. We find that the interference not only increases the repair time by 3.6–91.5%, but also prolongs the P99 latency of YCSB requests by 4.7–31.5%. The reasons are two-fold. On one hand, the dynamic foreground traffic makes the occupied bandwidth fluctuate over time (e.g., from 0.2 Gb/s to 3.6 Gb/s in a 10 Gb/s link). The original repair plans are unaware of bandwidth changes and hence cannot timely take actions to avoid bandwidth contentions. On the other hand, the utilization of link bandwidth is unbalanced in repair, where the most-loaded uplink supplies 110.5% of additional bandwidth compared to the least-loaded uplink (Section II-D). Hence, how to schedule repair in the face of the dynamic foreground traffic is a critical issue for promising system reliability and service quality.

We propose ChameleonEC, a new repair mechanism that exploits the tunability of erasure coding to achieve low-interference repair. To balance the bandwidth utilization, it decomposes a *repair plan* (including requested chunks and their transmission paths) into multiple *repair tasks*. Each repair task is responsible for uploading or downloading a requested chunk in repair. It dispatches the repair tasks across nodes according to the residual link bandwidth (i.e., leaving out the bandwidth occupied by the foreground traffic). It then establishes a tunable repair plan over the tasks based on the linearity and addition associativity of erasure coding to instruct the repair. ChameleonEC further monitors the repair progress and timely re-tunes data transmissions and repair plans to mitigate the repair stalls caused by unexpected stragglers.

Because of the tunability, ChameleonEC is a general design to achieve low-interference repair: (i) it can assist a variety of erasure codes for repair acceleration; (ii) it can achieve low-interference repair under the traces with different access

[1]Yuhui Cai and Shiyao Lin are co-first authors of this paper.

characteristics; (iii) it can accelerate both the single-node and the multi-node repairs; and (iv) it can be extended to the storage-bottlenecked scenarios (Section V-B). By distributing repair tasks according to the idle bandwidth, ChameleonEC can accelerate repair for both network-based storage systems (e.g., Ceph [79], Hadoop HDFS [9], and OpenStack [17]) and locally-attached disk array (e.g., declustered RAID [16]). To summarize, this paper makes the following contributions:

- We find that the interference between data repair and foreground traffic will increase the repair time and the latency of foreground requests (Section II-D).
- We propose ChameleonEC, an approach that exploits the tunability of erasure coding for low-interference repair. ChameleonEC carefully tunes repair plans to preempt the unoccupied bandwidth and re-schedules data transmissions to bypass emerging stragglers (Section III).
- We implement a ChameleonEC prototype with C++ and integrate it into Hadoop HDFS 3.1.4 (with around 260 LoC added) (Section IV).
- We conduct extensive experiments on Amazon EC2 [12], showing that when running with foreground requests, ChameleonEC can still aid a variety of erasure codes to accelerate the repair by 4.9–498.2% and reduce 8.4% of the P99 latency of foreground requests (Section V).

The source code of ChameleonEC can be reached via https://github.com/shenzr/ChameleonEC.

II. Background

A. Erasure Coding

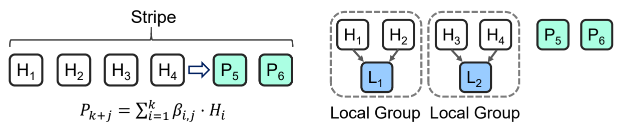
Erasure coding is usually defined via two integer parameters (namely k and m) to configure the fault tolerance degree and the storage overhead. Specifically, it encodes a collection of k fixed-size data chunks at a time to generate additional m *parity chunks*, where each parity chunk is calculated as a linear combination of the k data chunks over Galois Field arithmetic [63]; such $k+m$ chunks encoded together collectively constitute a coding group called *stripe*, which can restore the k original data chunks in the presence of no more than m chunk failures.

When a chunk C' fails, erasure coding will take back k surviving chunks (denoted by $\{C_1, C_2, \cdots, C_k\}$) and regenerate (decode) C' via the following equation:

$$C' = \sum_{i=1}^{k} \alpha_i C_i, \tag{1}$$

where α_i ($1 \leq i \leq k$) is the decoding coefficient assigned to the surviving chunk C_i for repairing C'. Hence, erasure coding tolerates any m chunk failures at the *storage amplification* of $\frac{k+m}{k}$, attaining the *maximum distance separable* (MDS) property [49]; furthermore, we can distribute the $k+m$ chunks of each stripe across $k+m$ different nodes (one chunk per node) to against at most m concurrent node failures.

There are various constructions of erasure codes [31], [35], [54], [58], [63], where Reed-Solomon (RS) codes [63] are the most popular paradigm used in production systems [6], [8], [13], [19], [51], [53] because they are *general* (i.e., it



(a) A stripe of RS(4,2). (b) A stripe of LRC(4,2,2).

Fig. 1. Examples of RS codes ($k = 4$ and $m = 2$) and LRCs ($k = 4$, $l = 2$, and $m = 2$), where H_i, $\beta_{i,j}$, and P_{k+j} denote the data chunk ($1 \leq i \leq k$), the encoding coefficient, and the parity chunk ($1 \leq j \leq m$), respectively. L_i is the local parity chunk in LRCs ($1 \leq i \leq l$).

can be configured by any k and m) and *systematic* (i.e., it keeps the k original data chunks untouched after encoding for efficient data accesses). For simplicity, we use RS(k, m) to denote the RS codes configured under k and m throughout the paper. Figure 1(a) shows a stripe of RS$(4, 2)$, where four data chunks (i.e., $\{H_1, H_2, \cdots, H_4\}$) are encoded to generate two additional parity chunks (i.e., $\{P_5, P_6\}$).

B. Repair in Erasure Coding

Failure operations: Failures in storage systems are often classified into (i) degraded reads and (ii) full-node repair. Degraded reads request the chunks that are temporarily unavailable (e.g., caused by network disconnections) and need to perform the repair on-the-fly, while full-node repair restores all the lost chunks of an entire node (e.g., caused by disk crashes). In this paper, we mainly focus on single-node repair, as it is one of the predominant failure events in practice [48], [62], [68]. We also evaluate the performance of degraded reads (Exp#10) and multi-node repair (Exp#8) in Section V-B.

Repair bottleneck: Repairing a single chunk in erasure coding requires two operations: (i) retrieving multiple surviving chunks (e.g., k chunks in RS(k, m)) over networks and (ii) restoring (decoding) the failed chunk (see Equation (1)). Existing studies in erasure coding often treat network bandwidth as the major bottleneck in repair, mainly due to the following reasons. First, the decoding bandwidth provided by modern CPU processors typically exceeds the network bandwidth of high-end NICs, making the decoding time negligible. For instance, the Intel Broadwell E5 v2680 can achieve a data decoding rate of 27.9 GB/s [69], which surpasses the 12.5 GB/s network bandwidth offered by an RDMA-enabled Mellanox EDR ConnectX-4 NIC. Previous research has also shown that decoding accounts for only 7.1% of the total repair time [50], [68]. Second, even in many high-performance network systems, the network bandwidth can be easily surpassed by the aggregated storage bandwidth [5], [39]. For example, EMC XtremIO can saturate its 80 Gb/s network bandwidth by using four out of the 18$\sim$72 high-end SSDs [39]. Hence, we first introduce our design in the network-bottlenecked scenarios [32], [45], [46], [48], [62], [68]. Compared to existing studies, we further show that our work is effective even in the storage-bottlenecked scenarios (see Exp#12 in Section V-B).

Impact of repair efficiency: We study the impact of repair efficiency on system reliability by measuring the changes of the data loss probability [66] under varied repair throughput (i.e., the amount of data repaired per time unit) in a single-

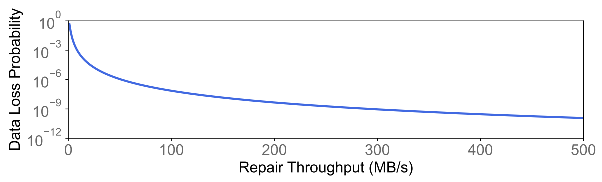


Fig. 2. Data loss probability under different repair throughput.

node repair. Our analysis can also reflect the trend of MTTDL [34] under different repair throughput.

Let θ be the expected lifetime of a node and suppose that the lifetime is exponentially distributed. This assumption enables us to calculate the probability that a node fails for a duration of τ [43], denoted as $f = 1 - e^{-\frac{\tau}{\theta}}$. We set θ to 10 years based on field studies [23] and estimate f under different durations.

For simplicity, we consider a storage system with $k + m$ nodes and employ RS(k, m) for fault tolerance. Each node stores exactly one chunk of a stripe and hence the system can tolerate any m node failures. When a node fails, let E_i denote the event where any additional i node failures occur during the single-node repair process, where $0 \le i \le k+m-1$. The probability of E_i (denoted by $\Pr(E_i)$) can then be calculated as $\Pr(E_i) = \binom{k+m-1}{i} \cdot f^i \cdot (1-f)^{k+m-1-i}$.

As the storage system can tolerate any m concurrent node failures, the data loss probability during the single-node repair can be computed as

$$\Pr_{\text{dl}} = 1 - \sum_{i=0}^{m-1} \Pr(E_i) \tag{2}$$

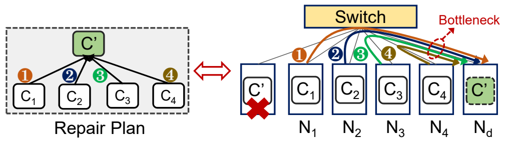
We finally study the change of $\Pr_{\text{dl}}$ under different repair throughput. We set k to 10 and m to 4, and assume that each node stores 96 TB data [44]. Figure 2 shows the analysis results, indicating that a higher repair throughput (i.e., a shorter repair procedure) will result in a lower $\Pr_{\text{dl}}$.

C. Existing Efforts in Erasure Coding

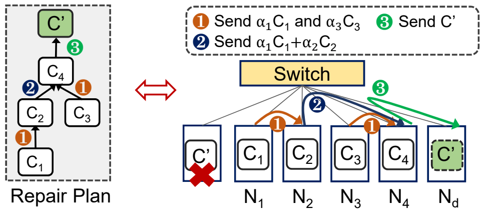
To accelerate repair in network-bottlenecked scenarios, extensive studies mainly focus on (i) constructing new repair-efficient codes that provably reduce repair traffic in coding theory and (ii) devising repair algorithms that parallelize data transmissions over networks for shortening repair latency.

Repair-efficient codes: They construct new families of erasure codes under different methodologies, including (i) increasing additional storage consumption (e.g., Locally Repairable Codes (LRCs) [29], [35], [40], [64] and Rotated Reed-Solomon (Rotated RS) codes [38]), (ii) requiring surviving nodes to send linear combinations of the locally stored data (e.g., PMSR [33]), (iii) allowing more nodes to participate in the repair (e.g., regenerating codes [25], [31], [54], [58]), and (iv) riding on the stripes of existing codes to create their dependency (e.g., Hitchhiker [62]).

Figure 1(b) shows a stripe of LRCs [35] defined via three parameters, namely k, l, and m. As a variant of RS(k, m), LRC(k, l, m) classifies the k data chunks of a stripe of RS(k, m) into l *local groups* (i.e., with $\frac{k}{l}$ data chunks per group when k is divisible by l) and maintains one additional *local parity chunk* for each group (e.g., L_1 and L_2 in Figure 1(b)). Hence, repairing a single data chunk in LRC(k, l, m) merely retrieves $\frac{k}{l}$ surviving chunks within the same group [2], as opposed to k chunks in RS(k, m). For example, LRC$(4, 2, 2)$ only takes back two chunks (H_2 and L_1) to repair H_1 in Figure 1(b).



(a) Conventional repair (CR) repairs C' in four timeslots.



(b) Partial-parallel-repair (PPR) [50] repairs C' in three timeslots.

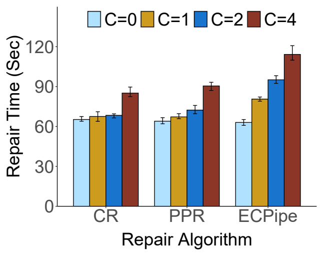
Fig. 3. Repairing C' under CR and PPR [50].

Repair algorithms: While repair-efficient codes reduce the amount of repair traffic in theory, repair algorithms further accelerate repair via exploiting transmission parallelism. The main idea is to decompose a chunk's repair into multiple sub-stages, which can be executed in parallel based on the linearity and addition associativity of erasure codes (see Equation (1)). Specifically, given a failed chunk of RS(k, m), a repair algorithm can generate a *repair plan* over $k+1$ nodes, including k *sources* (i.e., the nodes supplying surviving chunks for repair) and a *destination* (i.e., the node storing the chunk finally repaired). The repair plan describes the *transmission paths* and the *execution orders* among the $k+1$ participating nodes; if a node is required to relay multiple surviving chunks to repair the same chunk, it can combine the received chunks with the locally stored chunk into a *partially decoded chunk* (i.e., the linear combination of the input chunks based on Equation (1)) and forward it to the next node guided by the plan.

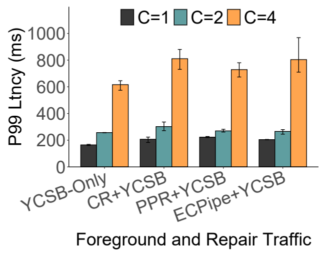
Figure 3 depicts two plans that repair the same chunk C' over four sources ($\{N_1, N_2, \cdots, N_4\}$) and a destination ($N_d$). For simplicity, we assume that it takes a *timeslot* to transmit a chunk from one node to another node over networks. Figure 3(a) shows a repair plan under *conventional repair* (CR). It directly transmits four surviving chunks to the destination N_d, making the downlink of N_d become the bottleneck. Finally, CR needs four timeslots to repair a chunk.

Figure 3(b) shows another repair plan under *partial-parallel-repair* (PPR) [50], whose routings follow a binary-tree structure and repair C' in three timeslots: (i) it first sends $\alpha_1 C_1$ from N_1 to N_2, while at the same time transmitting $\alpha_3 C_3$ from N_3 to N_4 (Step ❶); (ii) it then calculates the linear combination $\alpha_1 C_1 + \alpha_2 C_2$ on N_2 (i.e., the partially decoded chunk based on Equation (1)), which is then delivered from N_2 to N_4 (Step ❷); and (iii) it restores C' on N_4 (calculated

[2]Repairing a global parity chunk (i.e., P_{k+j} for $1 \le j \le m$) still needs to retrieve k chunks.



(a) Repair time.



(b) P99 latency of YCSB-A.

Fig. 4. A trace-driven analysis on repair time and access latency.

as $C' = \sum_{i=1}^{4} \alpha_i C_i$) and transmits it to N_d (Step ❸). Hence, given the same repair traffic (i.e., four chunks in this example), different repair plans may have different repair times.

D. *Existence of Interference and Root Causes*

For the first time, we study the interference between the repair traffic and the foreground traffic in erasure-coded storage. We select three repair algorithms: (i) CR (Figure 3(a)); (ii) PPR [50] (Figure 3(b)); and (iii) ECPipe [45], which partitions a chunk into multiple slices and pipelines their repair to approach $O(1)$ repair time. For a fair comparison, we also break a chunk into multiple fixed-size slices and pipeline their transmissions for CR and PPR.

We provision 20 instances on Amazon EC2 [12] and execute each repair algorithm to recover 200 chunks (each 64 MB in size). Concurrently, we initiate several clients to replay Yahoo! Cloud Serving Benchmark (YCSB-A) trace [24], which is of a 50% read and 50% update workload under the Zipfian distribution (the value size is 512 KB). We also evaluate more traces in Section V-B. We vary the number of clients (C) from zero (indicating no interference) to four (representing tremendous foreground traffic), and measure the repair time and P99 latency (tail latency at the 99^{th} percentile) of the YCSB requests (see Section V-A for experimental details).

Observations: Figure 4(a) indicates that the interference between data repair and foreground traffic increases the repair time by 3.6–91.5%, and the repair time increases with the number of clients issuing YCSB requests. Similarly, Figure 4(b) shows that the interference also lengthens the P99 latency of foreground traffic by 4.7–31.5% compared to that with no interference (i.e., YCSB-Only). In our experiments, CR outperforms PPR and ECPipe, which is contrary to the previous study [48]. We attribute this to the transmission dependency in the single-chunk repair, which makes PPR and ECPipe susceptible to network fluctuations. We further identify the root causes of the tight contentions between data repair and foreground traffic.

Root cause 1 (R1): The bandwidth occupied by the foreground traffic fluctuates. To learn the fluctuation of the used foreground bandwidth, we partition the trace duration into consecutive and non-overlapping *time windows*, where each time window spans 15 seconds [76]. For each node, we monitor the fluctuation (i.e., the maximum minus the minimum) of the bandwidth occupied by the foreground traffic

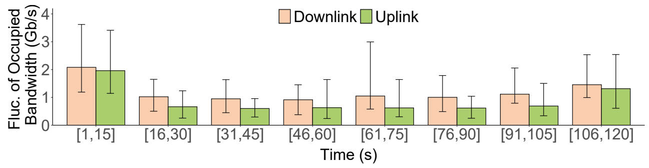
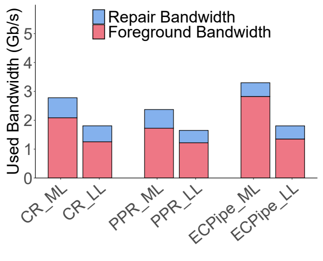
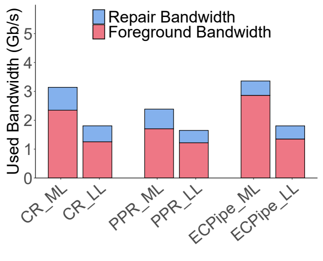


Fig. 5. The fluctuation of the bandwidth occupied by the foreground traffic.



(a) Used bandwidth of uplinks.



(b) Used bandwidth of downlinks.

Fig. 6. The bandwidth utilizations of the most-loaded (ML) and the least-loaded (LL) uplinks and downlinks.

in both uplinks and downlinks [3] in each time window, and plot the average as well as the maximum and minimum fluctuations of all nodes.

Figure 5 shows that the link bandwidth occupied by the foreground traffic keeps fluctuating, which changes by 1.1 Gb/s on average per time window and can reach up to 3.6 Gb/s. However, existing repair algorithms [45], [48], [50], [67], [68] do not consider the co-existence with the foreground traffic and are unable to promptly respond to bandwidth changes.

Root cause 2 (R2): The bandwidth utilization is unbalanced. We further analyze the bandwidth utilization of nodes. We measure the *repair bandwidth* and *foreground bandwidth* of the uplink and downlink for each node which denote the average bandwidth occupied by repair traffic and foreground traffic, respectively. We then sum up the repair and foreground bandwidth and get the most-loaded (ML) and the least-loaded (LL) uplinks and downlinks, respectively.

Figure 6 shows that the bandwidth utilization of uplinks and downlinks significantly varies across nodes. For instance, in ECPipe, the most-loaded uplink (ECPipe_ML in Figure 6(a)) supplies 110.5% of additional bandwidth compared to the least-loaded uplink (ECPipe_LL in Figure 6(a)). The underlying reasons are two-fold. First, existing repair algorithms cannot quickly adjust the repair plans when the foreground bandwidth changes (see R1); hence, the nodes serving a surge of foreground requests may have to process a large amount of repair traffic again. Secondly, existing repair algorithms typically restore lost chunks using fixed transmission paths (e.g., the binary-tree structure in PPR [50]), which are inflexible in adapting to bandwidth fluctuations.

III. ChameleonEC Design

We design ChameleonEC, an approach to achieve low-interference repair for erasure-coded storage. For each failed chunk, ChameleonEC divides its repair plan into multiple upload and download tasks (termed "repair tasks"), where

[3]We use `NetHogs` [14] to monitor the network bandwidth used by the YCSB requests.

each repair task handles the uploading or downloading of a chunk. ChameleonEC periodically assesses the link bandwidth to understand bandwidth heterogeneity, and distributes repair tasks across nodes to fully saturate available bandwidth (Section III-A, R2 addressed). It then establishes a tunable repair plan based on task distribution to guide repairs (Section III-B). As the foreground traffic fluctuates over time, ChameleonEC monitors repair progress regularly and dynamically re-tunes the repair plan (in repairing a single chunk) and re-schedules the transmission orders (in repairing multiple chunks), relieving repair stalls caused by emerging stragglers (Section III-C, R1 addressed).

A. Dispatching Repair Tasks

Main idea: To co-exist with the dynamic foreground traffic, ChameleonEC decomposes the entire repair process into multiple *repair phases* with a constant time span T_{phase}, such that it can react to bandwidth changes in a timely manner. In each phase, it repairs a few failed chunks based on the available link bandwidth, ensuring that the repair can be completed in this phase with a high probability. ChameleonEC repeats the repair phases until all the failed chunks are repaired.

Problem formulation: We first formulate the repair problem. Let $\overline{\mathcal{C}} = \{C_1, C_2, \cdots, C_f\}$ be a set of f failed chunks ($f \geq 1$). In each phase, we continuously select a failed chunk from $\overline{\mathcal{C}}$ one at a time to repair before the accumulated repair time exceeds T_{phase}. Repairing a chunk requires establishing a repair plan over $k+1$ nodes, with k sources supplying surviving chunks and a destination storing the repaired chunk (Section II-C). To saturate the available bandwidth, ChameleonEC decomposes a repair plan into k *upload tasks* (denoted by $\{U_1, U_2, \cdots, U_k\}$) and another k *download tasks* (denoted by $\{D_1, D_2, \cdots, D_k\}$), where each upload (resp. download) task is in charge of uploading (resp. downloading) a chunk for repair. For example, Figure 7 shows that the repair that retrieves four chunks (i.e., $k = 4$) can be decomposed into eight repair tasks ($\{U_1, U_2, \cdots, U_4\}$ and $\{D_1, D_2, \cdots, D_4\}$, Step ❶). ChameleonEC then seeks $k+1$ nodes to dispatch the $2k$ tasks to saturate the available bandwidth for repair.

Selecting a destination: ChameleonEC starts a chunk's repair with the selection of its destination. Given a failed chunk C' to be repaired, it first gets a set of candidate destinations (denoted by $\mathcal{D}$), satisfying that they do not store any chunk of the stripe to which C' belongs before repair. Repairing C' in a node selected from $\mathcal{D}$ ensures that this repaired stripe still spans across $k+m$ nodes to tolerate any m node failures (i.e., the fault tolerance is not compromised after repair, Section II-A).

For each node $N_i \in \mathcal{D}$, let T^i_{down} be the number of download tasks assigned to N_i at present, and B^i_{down} be the idle bandwidth of the downlink of N_i that can be learned by either periodically monitoring or pre-limiting by the system [76]. We choose the node N_d, which is estimated to take the minimum time to finish the download tasks after being assigned with a new download task (i.e., having the smallest $\frac{(T^i_{\text{down}}+1)\cdot|C|}{B^i_{\text{down}}}$, where $|C|$ is the chunk size), to serve as the destination. We

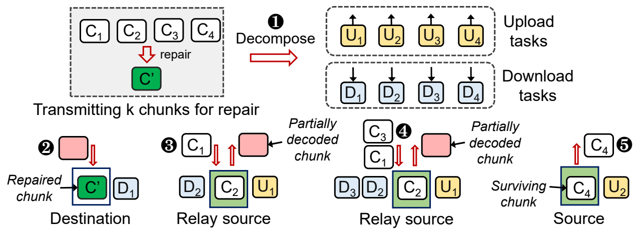


Fig. 7. The repair tasks and their functionalities ($k = 4$).

call it the *minimum-time-first selection*, which aims to balance the download time across nodes with a high probability.

Dispatching repair tasks: Besides $\mathcal{D}$, ChameleonEC also gets a set of $k+m-1$ candidate sources (denoted by $\mathcal{S}$), which store the surviving chunks within the same stripe of the failed chunk C'. ChameleonEC then seeks to dispatch the k upload tasks and another k download ones across the $k+m$ selected nodes (i.e., $\mathcal{S} \cup \{N_d\}$). At the very beginning, as the destination N_d has to download at least one chunk to repair C' (e.g., Step ❷ in Figure 7), ChameleonEC increases the number of its download tasks by one (i.e., $T^d_{\text{down}} = T^d_{\text{down}}+1$). ChameleonEC then enumerates each node N_i in $\mathcal{S} \cup \{N_d\}$ to assign the remaining $k-1$ download tasks. We consider the following two possibilities.

On the one hand, if we still choose to dispatch one additional download task to N_d (i.e., tentatively increase T^d_{down} by one), then the resulting repair time of N_d (denoted by R_d) in this phase can be estimated as

$$R_d = \max\{\underbrace{\frac{T^d_{\text{up}}\cdot|C|}{B^d_{\text{up}}}}_{\text{estimated upload time}}, \underbrace{\frac{(T^d_{\text{down}}+1)\cdot|C|}{B^d_{\text{down}}}}_{\text{estimated download time}}\},$$

where B^d_{up} and T^d_{up} are the idle link bandwidth of N_d and the number of upload tasks to be executed by N_d in this phase, respectively.

On the other hand, if we choose to dispatch a download task to a candidate source $N_i \in \mathcal{S}$, then N_i is supposed to act as a relay source in the repair of C' (e.g., Step ❸ in Figure 7). Different from the other sources that simply upload surviving chunks for repair (e.g., Step ❺ in Figure 7), a relay source will first combine all the chunks it receives (e.g., C_1 in Figure 7, see Step ❸) for repairing C', together with the locally stored chunk (e.g., C_2 in Figure 7, see Step ❸), to generate a partially decoded chunk. It then sends (uploads) this chunk to the next node to continue the repair. Consequently, assigning a download task to a candidate source N_i (increasing T^i_{down} by one) may also change the number of upload tasks it affords in the following two possibilities. If N_i has not been given any download task to repair C' at present, then assigning a download task to it (e.g., D_2 in Figure 7) will come with another associated upload task for uploading the partially decoded chunk (e.g., U_1 in Figure 7), and hence the repair time of N_i can be estimated as $R_i = \max\{\frac{(T^i_{\text{up}}+1)\cdot|C|}{B^i_{\text{up}}}, \frac{(T^i_{\text{down}}+1)\cdot|C|}{B^i_{\text{down}}}\}$ (see Step ❸ in Figure 8). Otherwise, if N_i has already been assigned an upload task in repairing C' (i.e., it is supposed to compute and upload a partially decoded chunk in repair),

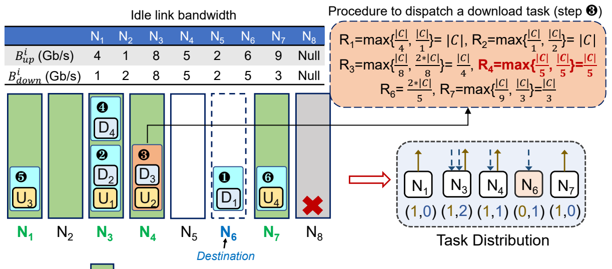

Idle link bandwidth
N1 N2 N3 N4 N5 N6 N7 N8
B_{up}^i (Gb/s) 4 1 8 5 2 6 9 Null
B_{down}^i (Gb/s) 1 2 8 5 2 5 3 Null
Procedure to dispatch a download task (step ❸)
R_1=max{|C|/4, |C|/1}= |C|, R_2=max{|C|/1, |C|/2}= |C|
R_3=max{|C|/8, 2*|C|/8}= |C|/4, R_4=max{|C|/5, |C|/5}=|C|/5
R_6= 2*|C|/5, R_7=max{|C|/9, |C|/3}=|C|/3
❹ D4
❷ D2 U1
❸ D3 U2
❶ D1
❺ U3
❻ U4
N1 N2 N3 N4 N5 N6 N7 N8
Destination
N1 N3 N4 N6 N7
(1,0) (1,2) (1,1) (0,1) (1,0)
Task Distribution
Candidate source
(u,d) # of upload and download tasks

Fig. 8. Example of assigning the upload and download tasks to repair a chunk, where $k = 4$ and $m = 2$.

adding an additional downloaded chunk (e.g., C_3 in Step ❹) to N_i will just update its partially decoded chunk without altering the number of upload tasks (e.g., combining C_1 and C_3 with C_2 to generate the partially decoded chunk, see Step ❹ in Figure 7), and hence the expected repair time of N_i can be calculated as $R_i = \max\{\frac{T_{\text{up}}^i \cdot |C|}{B_{\text{up}}^i}, \frac{(T_{\text{down}}^i + 1)\cdot|C|}{B_{\text{down}}^i}\}$.

ChameleonEC finally dispatches a download task to the node in $\mathcal{S} \cup \{N_d\}$ that is estimated to achieve the minimum repair time after the assignment. It repeats the selection until all the k download tasks have been assigned. For the remaining upload tasks (recall that some have been assigned together with the download ones), ChameleonEC chooses the sources that have no download tasks by following the minimum-time-first selection methodology, and each source can be assigned with at most one upload task, as it has only one surviving chunk for repairing C'.

After all the $2k$ tasks to repair C' have been dispatched, ChameleonEC then confirms whether the repair of C' can be completed within this phase, via checking the repair time of the selected sources and destination. If the estimated repair time exceeds the time span of the phase (i.e., T_{phase}), then ChameleonEC will try the repair of C' in the next phase.

Example: Figure 8 shows an example of dispatching repair tasks (where $k = 4$ and $m = 2$). It first distributes D_1 to the destination N_6 (i.e., $N_d = N_6$) that does not store any chunk of the same stripe before repair (Step ❶). It then assigns D_2 (together with U_1) to N_3 (Step ❷), and dispatches D_3 (together with U_2) to N_4 (Step ❸). After that, it assigns D_4 to N_3 again (Step ❹). Since N_3 already has an upload task (i.e., U_1), assigning D_4 does not introduce any additional upload task to N_3. It finally distributes U_3 and U_4 to the sources N_1 and N_7, respectively (Steps ❺ and ❻). After the assignment, we can get the task distribution across four selected sources $\{N_1, N_3, N_4, N_7\}$ and a destination N_6. Figure 8 also explains the procedure that uses idle bandwidth to perform Step ❸.

B. Establishing Tunable Repair Plans

After assigning the $2k$ repair tasks across the $k+1$ selected nodes, ChameleonEC then establishes a repair plan by pairing the upload and download tasks (i.e., determining the transmission path), so as to navigate the repair.

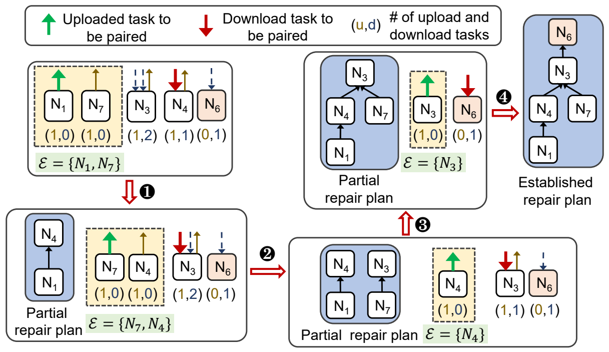
Without loss of generality, let $\mathcal{N} = \{N_1, N_2, \cdots, N_k, N_d\}$ be the $k + 1$ nodes chosen to repair the failed chunk C', where $\{N_1, N_2, \cdots, N_k\}$ are the sources while N_d denotes the destination. For each node $N_i \in \mathcal{N}$, we use a tuple $(T_{\text{up}}^i, T_{\text{down}}^i)$ to represent the numbers of upload and download tasks assigned to N_i for repairing C', respectively. For the sources, we can readily deduce that $T_{\text{up}}^i = 1$ for $1 \le i \le k$ (as each source has to upload exactly one chunk); while for the destination, we can get that $T_{\text{up}}^d = 0$ (as the destination finally recovers the failed chunk without needing to upload any chunk) and $T_{\text{down}}^d \ge 1$ (as the destination needs to download at least a chunk to complete the repair). Algorithm 1 presents the pseudo-code to establish a tunable repair plan based on the task distribution, whose main idea is to establish the transmission paths among the sources at first and finally complement the plan by adding the destination.

Algorithm 1 Establishing a tunable repair plan

Input: $\mathcal{N} = \{N_1, N_2, \cdots, N_k, N_d\}$ (selected k+1 nodes for repair)
Output: A repair plan

```
1:  procedure MAIN($\mathcal{N}$)
2:      Establish $\mathcal{E}$
3:      // Establish transmission paths among the sources
4:      while $\sum_{i=1}^{k} T_{\text{down}}^i \neq 0$ do
5:          Get $N_y$, where $T_{\text{down}}^y = \min\{T_{\text{down}}^i | T_{\text{down}}^i > 0, 1 \le i \le k\}$
6:          $N_x$=POP($\mathcal{E}$)
7:          Establish a transmission path from $N_x$ to $N_y$
8:          $T_{\text{up}}^x = T_{\text{up}}^x - 1$, $T_{\text{down}}^y = T_{\text{down}}^y - 1$
9:          Update $\mathcal{E}$
10:     end while
11:     // Establish transmission paths to the destination
12:     while $\mathcal{E} \neq \emptyset$ do
13:         Select $N_x \in \mathcal{E}$
14:         Establish a transmission path from $N_x$ to $N_d$
15:         $T_{\text{up}}^x = T_{\text{up}}^x - 1$ and update $\mathcal{E}$
16:     end while
17: end procedure
```

Algorithm details: ChameleonEC iteratively establishes the transmission paths via pairing the upload and download tasks. It first selects a set of nodes (denoted by $\mathcal{E}$) that satisfy both of the following conditions: (i) each node in $\mathcal{E}$ has an upload task unpaired; and (ii) each node either has no download task or has its download tasks all paired (Line 2). To connect the upload and download tasks, ChameleonEC scans the sources and selects the one (denoted by N_y, where $1 \le y \le k$) that has the fewest unpaired download tasks (Line 5). It then picks out a node N_x (where $1 \le x \neq y \le k$) from $\mathcal{E}$ and establishes a transmission path connected from N_x to N_y, implying that a chunk uploaded (sent) by N_x will be downloaded (received) by N_y (Lines 6-7). ChameleonEC then updates the number of unpaired upload and download tasks on N_x and N_y, respectively (Line 8). We then update the set $\mathcal{E}$ (Line 9): (i) if T_{down}^y is zero (i.e., all the download tasks of N_y have been paired), ChameleonEC appends N_y to $\mathcal{E}$ (since N_y is a source that still has an unpaired upload task); and (ii) if T_{up}^x is zero (i.e., all the upload tasks of N_x have been paired), ChameleonEC then evicts N_x from $\mathcal{E}$. ChameleonEC repeats the pairing operations until $\sum_{i=0}^{k} T_{\text{down}}^i = 0$, indicating that all the download tasks assigned to the k sources have been paired (Lines 4-10). After establishing the transmission paths



Uploaded task to be paired
Download task to be paired
(u,d) # of upload and download tasks
$\mathcal{E} = \{N_1, N_7\}$
Partial repair plan
$\mathcal{E} = \{N_7, N_4\}$
Partial repair plan $\mathcal{E} = \{N_4\}$
Partial repair plan
$\mathcal{E} = \{N_3\}$
Established repair plan

Fig. 9. Example of establishing a chunk's repair plan.

among the k sources, ChameleonEC finally pairs the sources' remaining upload tasks with the destination's download tasks, hence generating the repair plan (Lines 12-16).

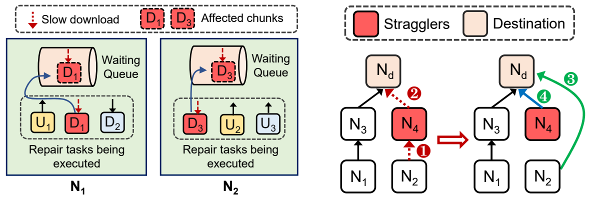
Example: Based on the task distribution in Figure 8, Figure 9 establishes a tunable repair plan to accomplish the repair. Given the four selected sources ($\{N_1, N_3, N_4, N_7\}$) in Figure 8, we first get $\mathcal{E} = \{N_1, N_7\}$, as they have no download task. We then find that N_4 has the fewest download tasks unpaired yet. We connect the upload task (marked in green) on N_1 and the download one (marked in red) on N_4 to generate a partial repair plan. We update $\mathcal{E} = \{N_7, N_4\}$, since the download tasks on N_4 have all been paired (Step ❶). We continue pairing the residual download tasks on N_3 (Steps ❷ and ❸). After all the download tasks on the sources are paired, ChameleonEC finally connects the residual upload task on N_3 with the download task of the destination (N_6), so as to generate the repair plan (Step ❹).

Complexity analysis: We note that establishing the set $\mathcal{E}$ (Line 2) and pinpointing the node N_y (Line 5) both introduce the complexity of $O(k)$. At most k transmission paths are to be determined among the sources, resulting in the complexity of $O(k^2)$ (Lines 2-10). Establishing the paths among the sources and the destination requires at most $O(k)$ trials (as a destination receives at most k chunks). Hence, the computational complexity of Algorithm 1 is $O(k^2)$.

C. Straggler-Aware Re-Scheduling

For each failed chunk, ChameleonEC selects the $k+1$ nodes to participate in the repair (Section III-A) and establishes a tunable repair plan (Section III-B). However, repair stragglers may appear unexpectedly, since the foreground traffic changes over time (R1 in Section II-D). To proceed with the repair even in the presence of the repair stragglers, ChameleonEC proposes a straggler-aware re-scheduling algorithm, whose main idea is to re-order the executions of repair tasks and re-tune repair plans, so as to timely bypass unexpected stragglers.

When generating repair plans, ChameleonEC assigns each repair task an *expectation*, which denotes the expected time to complete the task. ChameleonEC then distributes the repair tasks to the nodes along with their expectations. When the repair proceeds, ChameleonEC periodically checks the repair progress (i.e., the number of finished repair tasks) and confirms if the repair progress is in line with the expectations.



(a) Transmission re-ordering. (b) Repair re-tuning.

Fig. 10. Example of straggler-aware re-scheduling. In (a), the tasks with the same color are designated to repair the same chunk.

Transmission re-ordering: ChameleonEC can be aware of the presence of the stragglers once it finds that some repair tasks are delayed unexpectedly (i.e., their finish time has exceeded the pre-given expectation by a given time threshold). In this case, ChameleonEC will perform the re-scheduling via two approaches. The first is *transmission re-ordering*, which aims to prevent the delay of this repair task from propagating to other chunks' repair. Specifically, in the transmission re-ordering, ChameleonEC examines the unfinished repair tasks that cooperatively repair the same failed chunk with the delayed task, and postpones their transmissions. ChameleonEC then executes other repair tasks at first, puts the postponed ones into a *waiting queue*, and looks for an opportunity to wake them up in the rest of this phase: if a node has finished the transmissions of the assigned tasks except the postponed ones before the end of this phase, ChameleonEC will restart the transmission of the postponed tasks. If the postponed repair tasks are not all finished during this phase, their transmissions will be restarted opportunistically in the next phases.

Figure 10(a) shows an example. Once a download task D_1 (on N_1) is delayed, ChameleonEC will append it along with the associated task D_3 (on N_2), both of which are cooperatively repairing the same chunk, to the waiting queues, and execute other repair tasks at first. By doing so, the other chunks' repair will not be delayed by the stragglers.

Repair re-tuning: The transmission re-ordering algorithm is a *reactive approach* that defers the repair of chunks whose tasks are stalled by the stragglers. Additionally, ChameleonEC introduces a *proactive* algorithm for repair re-tuning. This algorithm adjusts the repair plans of affected chunks to bypass stragglers and ensure fast repair. Specifically, when identifying a straggler within a repair plan, ChameleonEC pinpoints the affected task. If the task involves downloading, ChameleonEC redirects the delayed download task to be executed at the destination. This adjustment offers two main advantages: (i) removing the delayed download task from the straggler, allowing normal repair to proceed; and (ii) enabling the dependent tasks (e.g., the upload task on N_4 in Figure 10(b)) to be executed without waiting for the download's completion, thereby accelerating the repair process. Currently, repair re-tuning only addresses delays in download tasks from the sources. Thus, we can use transmission re-ordering and repair re-tuning in tandem to effectively mitigate repair stalls caused by stragglers.

Figure 10(b) illustrates that the downlink bandwidth of the node N_4 is constrained, causing a delay in its download task (i.e., ❶). Consequently, the repair process is halted, as N_4 needs to merge the downloaded chunk with the locally stored chunk and then upload the combined result to the destination node (N_d) for repair (i.e., ❷). In such scenarios, ChameleonEC alters the repair plan by re-directing the delayed download task to the destination node (i.e., ❸), which eliminates the transmission dependency between N_2 and N_4 and ensures the proper executions of this download task and the upload task on N_4 (i.e., ❹).

D. Extensions

Repair for multi-node failure: ChameleonEC can be extended for multi-node repair by offering three options. First, it can repair each node one after another until all the failed nodes are successfully repaired. Second, it can assign the stripes that comprise more failed chunks with higher repair priorities and schedule the repair of the failed chunks based on their priorities. Third, it can repair chunks based on their estimated repair time and give the chunk with less repair time a higher repair priority. All three approaches have the same repair traffic, while the second concerns more on data reliability, and the third prioritizes repair efficiency.

Repair for different erasure codes: We demonstrate that ChameleonEC also works for other erasure codes. We take LRCs [35], [40], [71] as an example. The repair of LRCs is similar to RS codes except that it repairs a failed data chunk using the remaining $\frac{k}{l}$ chunks of the same local group (suppose that k is divisible by l, see Section II-B). Hence, ChameleonEC can simply dispatch the $\frac{k}{l}$ upload tasks and another $\frac{k}{l}$ download tasks to saturate the available bandwidth and establish a tunable repair plan as in Section III-B. We evaluate the performance of ChameleonEC for LRCs and Butterfly code [54] in Section V-B.

Repair in storage-bottlenecked scenarios: If the bottleneck shifts to the storage bandwidth, ChameleonEC can monitor the storage bandwidth consumption (as opposed to the network bandwidth consumption) and dispatch the read and write tasks across nodes. It then establishes the tunable repair plan based on the task distribution. We show the effectiveness of ChameleonEC in the storage-bottlenecked scenarios (see Exp#12 in Section V-B).

IV. Implementation

We implement a ChameleonEC prototype with around 3,000 lines of code (LoC) in C++. We employ Jerasure Library v2.0 [3] and GF-complete [56] for encoding and decoding. We employ Redis [4] for data storage across nodes.

System architecture: Figure 11 presents the system architecture of ChameleonEC, with a centralized *coordinator* and multiple *proxies*. The coordinator runs in the metadata server to access the metadata information (e.g., the stripe organizations and the chunk locations), while the proxies are implemented in storage nodes for data access. The coordinator and the proxies work cooperatively to accomplish the repair: (i) the coordinator first generates repair plans once receiving the

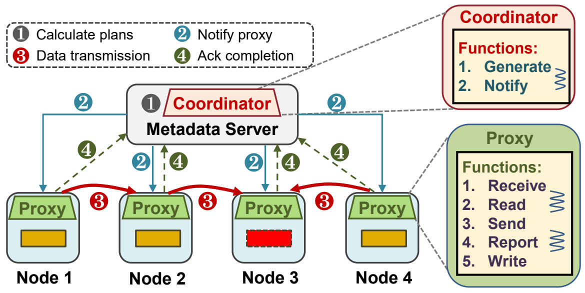


Fig. 11. The architecture of ChameleonEC.

alert of data loss (Step ❶), and notifies the corresponding proxies to start the repair (Step ❷); (ii) upon receiving the commands, the proxies can learn their missions by parsing the commands, each of which performs data accesses and transmissions (Step ❸); (iii) the proxies monitor the repair progress and adjust both transmission orders and repair plans if sensing that the repair progress turns slow; and (iv) the proxies finally send ACKs to the coordinator once the repair completes (Step ❹).

Integration to Hadoop HDFS: We integrate ChameleonEC into Hadoop HDFS 3.1.4 [9] by adding around 260 LoC [4]. We deploy the coordinator on NameNode and run the proxies on DataNodes. The raw HDFS maintains periodical heartbeat communications to indicate the liveness of DataNodes. Hence, when the heartbeat from a DataNode is absent for a long time, the NameNode will treat the DataNode as a "dead" node; at this time, ChameleonEC takes over the repair on behalf of HDFS by accessing the metadata information and generating repair plans for the chunks on the failed DataNode. We add the `ChameleonECReconstructor` class into the codebase of the DataNode. This class calls ChameleonEC to perform the repair and hands over the repaired chunks to the DataNode after the repair is completed. Finally, the DataNodes that store the repaired chunks report the new locations of these chunks to the NameNode through the heartbeats, which then updates the metadata information.

V. Evaluation

We conduct extensive testbed experiments on Amazon EC2 [12] and summarize our major findings as follows.

- ChameleonEC improves the repair throughput by 4.9–498.2% and reduces P99 latency by 8.4% (Exp#1–#13);
- ChameleonEC reduces the increase degree of trace execution times by 6.2–81.4% (Exp#2) and gains higher repair throughput under a smaller T_{phase} (Exp#3);
- ChameleonEC can promptly respond to trace changes and gain a high repair throughput (Exp#4);
- ChameleonEC generates repair plans quickly (Exp#5);
- ChameleonEC gains a higher repair throughput than the repair algorithms using RepairBoost [48] (Exp#6);
- ChameleonEC still improves the repair throughput by 25.0–41.3% with no foreground traffic (Exp#7);

[4]While HDFS adopts Intel ISA-L [2] for encoding, ChameleonEC successfully repair the failed chunks by Jerasure Library using the same Cauchy encoding matrix.

- ChameleonEC retains its effectiveness for multi-node repair (Exp#8) and various erasure codes (Exp#9);
- ChameleonEC can accelerate repair for degraded reads (Exp#10) and storage-bottlenecked scenarios (Exp#12);
- The design techniques in ChameleonEC are orthogonal and complementary to each other (Exp#11).

A. Experimental Setup

Testbed and preparations: To ensure the reproducibility of experimental results, we perform our experiments on Amazon EC2 [12], consistent with previous studies [26], [48], [77]. We implement security-group-level containment to guarantee isolation among instances. We provision 20 `m5.xlarge` instances in the region of North Virginia. Each instance runs Ubuntu 16.04.7 LTS and owns four vCPU with 3.1 GHz Intel Xeon Platinum processor, 16 GB RAM, and 300 GB General Purpose SSD volumes. The storage bandwidth is around 500 MB/s and the network bandwidth is 10 Gb/s.

Comparison approaches: We compare ChameleonEC to the following representative repair algorithms with different scheduling strategies: (i) CR, which directly reads the surviving chunks for repair without performing any scheduling; (ii) PPR [50], which constructs binary-tree-like transmission paths to enable transmission parallelism; and (iii) ECPipe [45], which constructs chained transmission paths for repair pipelining. We further use RepairBoost [48] for repair acceleration (see Exp#6). We measure a metric named *repair throughput*, defined as the amount of data being repaired per time unit (i.e., the inverse of the mean latency to repair per data size) [48], [67]. Generally, a higher repair throughput indicates that the lost data can be repaired in a shorter time window. We randomly select the k sources for the three repair algorithms, since the random selection can generate more balanced repair traffic in most cases than the LRU-based selection [48]. We partition a chunk into multiple smaller *slices*. This enables the pipelining of storage I/O (for reading and writing slices) and network I/O (for transmitting slices over network). We apply slicing to all repair algorithms for a fair comparison.

Default configurations: We set the chunk size to 64 MB (used in Hadoop HDFS [13]) and the slice size to 1 MB. We select RS$(10, 4)$ as the default erasure coding paradigm (used in Facebook f4 [51]). We set the length of a repair phase (i.e., T_{phase}) to 20 seconds. We repair 200 chunks in the full-node repair (with 125 GB of repair traffic) and repair a chunk in degraded reads. We repeat each experiment for five runs.

To mimic the foreground traffic, we run YCSB-A in four instances, each of which generates 100,000 requests (50% reads and 50% updates) with a 512 KB value size [21], [37]. The aggregated foreground traffic of YCSB is about 195.3 GB. When evaluating the repair performance, we use YCSB-A to generate the default foreground activities throughout the paper except Exp#1, Exp#2, and Exp#4.

B. Experimental Results

Exp#1 (Interference study): We first study the interference between the repair and foreground traffic. We replay the following traces with different access characteristics when the repair is running. We measure the P99 latency of each trace and the resulting repair throughput.

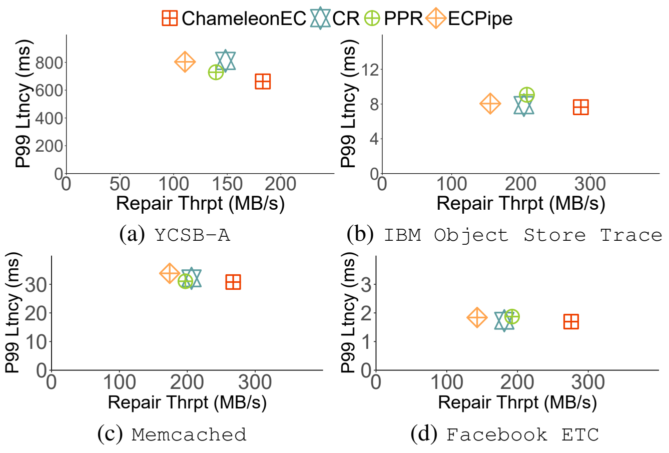


Fig. 12. Exp#1 (Interference study).

- `YCSB-A` on HBase [1]: It comprises 50% of reads and 50% of updates with 512 KB values. It issues 100,000 requests under the Zipfian distribution ($\alpha = 0.99$). This trace can assess the impact of repairs in read-write balanced scenarios.
- `IBM Object Store Trace` [10]: We select the trace with Number 000 from the IBM Cloud Object Storage Service [27], where the values dramatically vary from 16 B to 2.4 GB. We replay the first 300,000 requests. This selection enables us to evaluate the impact of repairs under significantly varied value sizes.
- `Memcached` [11]: It is collected from Twitter's in-memory caching with 54 clusters [81]. We select the trace from Cluster 37, which has the key size of 72 B and the value size of 20,134 B on average. It has 63% of GET operations and 37% of SET operations. This selection can help evaluate the impact of repairs under small value size.
- `Facebook's ETC workloads on Memcached` [18]: It has a GET/UPDATE ratio of 30:1. We generate keys and values according to the generalized extreme value distribution and Pareto distribution, respectively. This trace is to evaluate the impact of repairs in read-dominated scenarios.

Figure 12 indicates that compared to CR, PPR, and ECPipe, ChameleonEC improves the repair throughput by 23.5%, 31.4%, and 65.6% on average across different traces, respectively. In addition, ChameleonEC shortens 18.2%, 9.1%, and 17.6% of the P99 latency of the traces on average when compared to CR, PPR, and ECPipe, respectively. The reason is that ChameleonEC distributes repair tasks based on the available bandwidth of nodes (Section III-A) and relieves repair stalls caused by stragglers (Section III-C), effectively mitigating the interference with the foreground traffic during repair. Besides, we notice that ECPipe achieves the lowest repair throughput, since it establishes a chain-like repair plan, which has the most

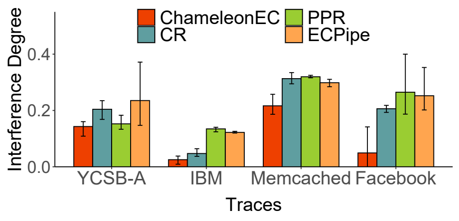


Fig. 13. Exp#2 (Impact on trace execution time).

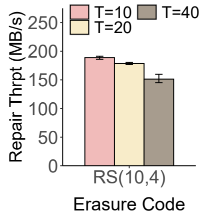


Fig. 14. Exp#3 (Impact of T_{phase}).

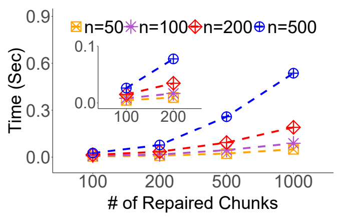


Fig. 16. Exp#5 (Computation time).

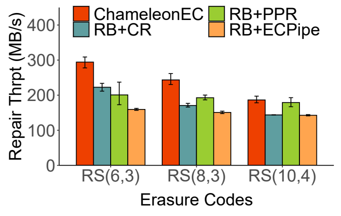


Fig. 17. Exp#6 (Performance under RepairBoost).

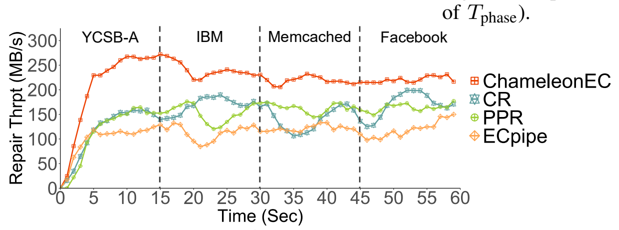


Fig. 15. Exp#4 (Adaptivity).

strict transmission dependency in data repair (i.e., a slice can be sent only after receiving its predecessors).

Exp#2 (Impact on trace execution time): Except measuring P99 latency in Exp#1, we introduce a metric called *interference degree* to quantify the increase ratio in the trace execution time. Given a trace, suppose that T is the trace execution time without repair and T^* denotes that under repair. Then the interference degree is defined as $\frac{T^*}{T} - 1$.

Figure 13 demonstrates that ChameleonEC reduces the interference degree by an average of 45.9%, 50.2%, and 56.7% compared to CR, PPR, and ECPipe, respectively. Notably, ChameleonEC introduces the least interference for traces with significantly changed values (e.g., IBM Object Store Traces [10]) and high heterogeneity (e.g., Facebook's ETC Workloads [18]), as it effectively distributes repair tasks to utilize unbalanced bandwidth to minimize interference.

Exp#3 (Impact of T_{phase}): We study the performance of ChameleonEC when the length of the repair phase (i.e., T_{phase}) varies. We consider RS$(10, 4)$ that is used in Facebook f4 [51] and change T_{phase} from 10 seconds to 40 seconds.

Figure 14 shows that as T_{phase} increases, the repair throughput of ChameleonEC gradually declines, as a smaller T_{phase} enables ChameleonEC to more timely react to network bandwidth fluctuations. However, frequently adjusting the distribution of repair tasks also comes with more management overhead, as ChameleonEC needs to keep monitoring the bandwidth changes. When T_{phase} is set to 20 seconds, the repair throughput of ChameleonEC only drops 5.4% compared to that when T_{phase} is 10 seconds. Hence, we suggest setting T_{phase} to 20 seconds for balancing management burden and repair performance.

Exp#4 (Adaptivity): We evaluated the repair performance under each individual trace in Exp#1. In this experiment, we extend the evaluation under dynamically transitioning traces. During repair, we replay each trace for 15 seconds, transition to another trace, and record the resulting repair throughput.

Figure 15 illustrates the fluctuations in repair throughput for different repair algorithms. ChameleonEC achieves an average improvement of 51.5%, 53.0%, and 97.2% across various traces compared to CR, PPR, and ECPipe, respectively. Additionally, when a new trace begins, ChameleonEC initially experiences a slight drop in repair throughput (e.g., an 18.9% decrease between 15 and 20 seconds) and then quickly regains its performance advantage. This drop occurs because ChameleonEC is initially unaware of the changed access characteristics but swiftly adapts by adjusting its strategy for distributing repair tasks. This experiment highlights ChameleonEC's good adaptability to dynamically fluctuating foreground traffic.

Exp#5 (Computation time): As ChameleonEC needs to dispatch repair tasks (Section III-A) and establish repair plans (Section III-B) according to the idle bandwidth, it calls for additional computation time compared to other repair algorithms. We measure the time by running ChameleonEC coordinator on one instance. Figure 16 shows the computation time versus the number of nodes (i.e., n) and the number of failed chunks. We make three observations.

First, when the number of nodes is fixed, the computation time of ChameleonEC generally increases with the number of chunks repaired in a phase, since ChameleonEC needs to generate a repair plan for each failed chunk. Second, to repair a constant number of failed chunks, ChameleonEC takes more computation time when deployed in larger storage systems, as ChameleonEC has to traverse more nodes to dispatch repair tasks. Third, most of the time, ChameleonEC merely needs no more than 0.6 seconds to generate repair plans; for instance, the computation time in a phase is 0.55 seconds when repairing 1,000 chunks for a 500-node storage system. This experiment demonstrates the good scalability of ChameleonEC, since the coordinator only conducts lightweight computations to generate repair plans and contacts proxies during repair.

Exp#6 (Performance under RepairBoost): RepairBoost (RB) [48] is a framework to boost the single-node repair for various repair algorithms. We assess the performance when CR, PPR, and ECPipe are all boosted under RepairBoost, denoted by RB+CR, RB+PPR, and RB+ECPipe, respectively.

First, CR, PPR, and ECPipe can achieve higher repair throughput under RB; for example, RB improves the repair throughput for ECPipe on average from 110.6 MB/s (Figure 12) to 142.7 MB/s (Figure 17) by balancing the repair traffic across nodes in the multi-chunk repair and scheduling the data transmission to maximize bandwidth utilization [48]. Second, compared to RB+CR, RB+PPR, and RB+ECPipe, ChameleonEC still improves the repair throughput by 34.8%, 16.7%, and 46.2%, respectively. Although RB can help repair

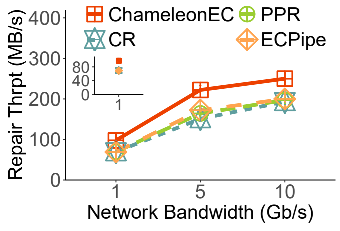


Fig. 18. Exp#7 (Performance with no foreground traffic).

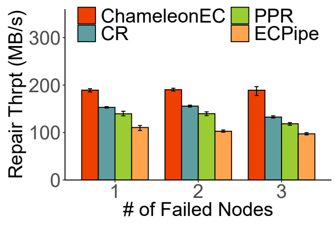


Fig. 19. Exp#8 (Multi-node repair).

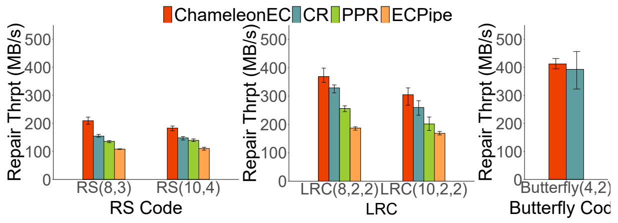


Fig. 20. Exp#9 (Generality).

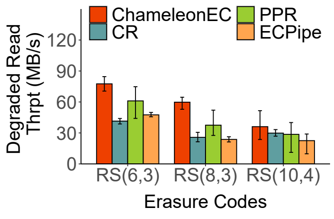


Fig. 21. Exp#10 (Degraded read).

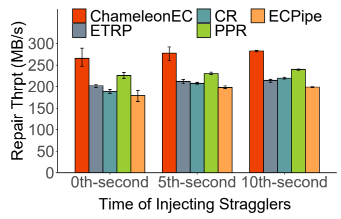


Fig. 22. Exp#11 (Breakdown study).

algorithms balance the repair traffic, the repair algorithms still have to establish the repair plans in a constant structure (e.g., the binary-tree structure in PPR [50]), reproducing the unbalanced bandwidth utilization again (see R2 in Section II-D). ChameleonEC can generate a tunable repair plan based on the unoccupied bandwidth, balancing the bandwidth utilization.

Exp#7 (Performance with no foreground traffic): We further measure the repair performance with no foreground traffic. We use `wondershaper` [15] to throttle the network bandwidth and vary the link bandwidth from 1 Gb/s to 10 Gb/s.

Figure 18 shows that running each repair algorithm with no foreground traffic generally gains higher repair throughput (e.g., 199.9 MB/s by ECPipe) than that when co-existing with the foreground traffic (e.g., 110.6 MB/s by ECPipe in Figure 24), since the repair is not interfered by the foreground traffic and can utilize more idle network bandwidth. Besides, ChameleonEC also achieves higher repair throughput (35.1% on average) than other repair algorithms across different network bandwidth, since it dispatches repair tasks across nodes based on the available bandwidth, which can effectively balance bandwidth utilization in multi-chunk repair. As a comparison, the other three algorithms select surviving nodes for repair by random (or using LRU algorithm [50]) without concerning bandwidth contentions.

Exp#8 (Multi-node repair): We evaluate the repair performance when the number of failed nodes increases from one to three and show the results in Figure 19.

With the number of failed nodes grows, the repair throughput of the algorithms slightly declines. The reasons are two-fold. First, with more failed nodes, the algorithms have fewer nodes to dispatch repair tasks for establishing repair plans. Second, the aggregated network/storage bandwidth shrinks once more nodes fail, hence limiting the repair parallelism. We also find that ChameleonEC still outperforms the other three repair algorithms in coping with multi-node failure. Specifically, ChameleonEC improves the repair throughput by 43.6% (for a single node failure) and by 65.7% (for triple node failures), since ChameleonEC performs better in scenarios with stringent bandwidth (see Exp#13).

Exp#9 (Generality): We demonstrate the generality of ChameleonEC for various erasure codes. We choose the following representative RS codes: (i) RS$(8,3)$ used in Yahoo! Cloud Object Store [7]; and (ii) RS$(10,4)$ used in Facebook f4 [51]. We then select the following practical LRCs: (i) LRC(8,2,2) evaluated in [40]; and (ii) LRC(10,2,2) considered in [65]. We also select a regenerating code named Butterfly code with $(n,k) = (4,2)$. To repair a failed chunk, Butterfly(4,2) simply retrieves multiple sub-chunks from different available chunks; therefore, we only consider ChameleonEC and CR for Butterfly(4,2).

Figure 20 shows that ChameleonEC outperforms other repair algorithms for RS codes and LRCs, which improves the repair throughput by 12.2–35.7%, 31.4–54.2%, and 65.7–97.0% compared to CR, PPR, and ECPipe, respectively. For regenerating codes, ChameleonEC slightly improves the repair throughput (by 4.9%) for Butterfly(4,2), since Butterfly(4,2) directly sends the surviving sub-chunks for repair and ChameleonEC cannot establish the elastic repair plan. Besides, for a repair algorithm, the repair throughput it gains for LRCs is significantly higher than that for RS codes, as given the same value of k, LRCs read fewer surviving chunks (within a local group) than RS code to repair a failed data chunk.

Exp#10 (Degraded read): We study the degraded read performance of ChameleonEC. We launch a client to request a chunk in a failed node and use each repair algorithm to restore the requested chunk. We measure the latency from the time of issuing the read request until this unavailable chunk is repaired [45]. We then compute the degraded read throughput.

Figure 21 shows that ChameleonEC improves the degraded read throughput by 20.9–152.0% under different erasure coding parameters. The improvement gained by ChameleonEC on the degraded read throughput gradually drops with the increase of k. For example, for RS$(6,3)$, ChameleonEC improves 59.1% of the repair throughput on average compared to the other three algorithms, while for RS$(10,4)$, the improvement reduces to 35.7%. This is because when k increases to 10, a chunk's repair will access half of the nodes in this testbed (our testbed has 20 nodes), hence narrowing the optimization space of ChameleonEC.

Exp#11 (Breakdown study): We perform a breakdown analysis via decomposing ChameleonEC into the following two components: (i) *establishing tunable repair plans* (ETRP), which distributes repair tasks based on the available bandwidth (Section III-A) and establishes a tunable repair plan based on the task distribution (Section III-B); and (ii) *straggler-aware*

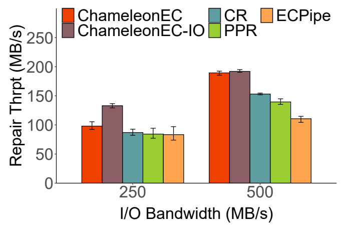


Fig. 23. Exp#12 (Performance in storage-bottlenecked scenarios).

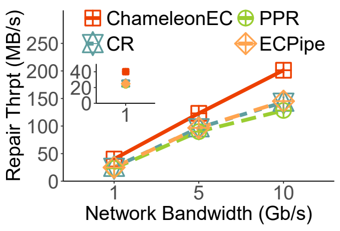


Fig. 24. Exp#13 (Impact of network bandwidth).

re-scheduling (SAR), which re-schedules repair progresses by adjusting transmission orders and re-tuning repair plans (Section III-C). To mimic a straggler, we launch a Redis client to run eight threads, each of which continuously reads 1-MB objects from a dedicated node participating in the repair. Since a repair phase spans 20 seconds, we start the straggler at different time points of a phase (the 0-second point, the 5-second point, and the 10-second point) and measure the repair throughput of different algorithms in this phase.

Figure 22 implies that when a straggler appears in a repair phase, ChameleonEC (i.e., ETRP+SAR) improves the repair throughput in this phase by 34.5%, 18.8%, and 43.5% compared to CR, PPR, and ECPipe, respectively. Although ETRP can dispatch repair tasks based on the available bandwidth, ChameleonEC still achieves higher repair throughput (31.4% on average) than ETRP. The reason is that ChameleonEC can timely adjust repair plans to bypass unexpected stragglers. We also find that each algorithm achieves higher repair throughput if the straggler appears later in the monitored phase, since fewer repair tasks in this phase are impacted by the straggler.

Exp#12 (Performance in storage-bottlenecked scenarios): When deploying ChameleonEC in the storage-bottlenecked scenarios, we can generate a variant named ChameleonEC-IO, which monitors the usage of the storage bandwidth and dispatches repair tasks based on the available storage bandwidth. We vary the storage bandwidth from 250 MB/s (i.e., $\frac{\text{Network bandwidth}}{\text{Storage bandwidth}} = 5$) to 500 MB/s (i.e., $\frac{\text{Network bandwidth}}{\text{Storage bandwidth}} = 2.5$). Figure 23 shows that the effectiveness of ChameleonEC reduces under stringent storage bandwidth, where the improvement of the repair throughput drops from 43.8% (when the storage bandwidth is 500 MB/s) to 15.5% (when the storage bandwidth is 250 MB/s), since the network bandwidth has less impact on the repair if the storage bandwidth becomes the new bottleneck. Moreover, ChameleonEC-IO outperforms ChameleonEC by improving 35.7% of the repair throughput when the storage bandwidth becomes stringent, since ChameleonEC-IO dispatches repair tasks based on the storage bandwidth consumption.

Exp#13 (Impact of network bandwidth): We finally study how ChameleonEC performs under different network bandwidths, which is varied from 1 Gb/s (evaluated in RAFI [28]) to 10 Gb/s (evaluated in ECWide [32] and OpenEC [47]).

Figure 24 shows that the repair throughput generally increases with the network bandwidth, since more data chunks can be transmitted over networks under the richer bandwidth. However, the performance improvement gained by ChameleonEC decreases as the network bandwidth increases, dropping from 64.4% at 1 Gb/s to 40.1% at 10 Gb/s, since once the storage I/O dominates the repair, the scheduling based on the network bandwidth utilization becomes less effective.

VI. Related Work

Repair-efficient codes: Regenerating codes [25] allow the selected sources to compute the linear combinations of the locally stored data before the transmission, which mainly differs in the fault tolerance (e.g., tolerating double failures [31], [54] and any m failures [41], [60], [72]), the size of the finite field used (e.g., the small-sized field [60], [72] and the large-sized field [41]), and the repair exactness (i.e., functional repair [25], [70] and exact repair [58], [73]). Butterfly code [54] and Clay code [74] further eliminate the local computations and send the locally stored data for repair directly. Geometric Partitioning [65] uses variable chunk sizes to achieve low degraded read time and high repair throughput. LRCs [35], [40], [71] and Rotated RS codes [38] reduce repair traffic with additional parity chunks. Hitchhiker [62] creates the dependency among stripes to decrease repair traffic. As a comparison, ChameleonEC schedules repair for various codes.

Repair algorithms: PUSH [36] pipelines the transmission of requested chunks to accomplish a full-node repair. CAR [68] and PPR [50] decompose a single-chunk repair into several independent sub-operations and parallelize their executions. ECPipe [45] breaks a chunk into many fixed-size sub-chunks and pipelines their repair to approach $O(1)$ time complexity. ClusterSR [67] balances the cross-rack repair traffic for the hierarchical data centers. RepairBoost [48] is a single-node repair framework to balance repair traffic and saturate the unoccupied bandwidth. Cocytus [82] repairs the requested data at first and then repairs the cold data when the system is idle. Existing studies ignore the interference with the foreground traffic, while ChameleonEC generates a tunable repair plan in response to the dynamic foreground traffic.

Some studies schedule the repair for replication-based storage. Dayu [76] monitors the foreground traffic and proactively adjusts the repair plan to relieve the interference. RAMCloud [52] scatters the repair tasks across the whole system to leverage the fruitful resource. In contrast, ChameleonEC is a low-interference repair mechanism for erasure-coded storage, which requires more considerations in repair, such as the selection of k sources and destination, and the scheduling of chunk transmissions for bandwidth utilization.

VII. Conclusions

We present ChameleonEC, a mechanism that exploits the tunability of erasure coding for low-interference repair. Its main idea is to carefully dispatch repair tasks across nodes based on the unoccupied bandwidth and establish a tunable repair plan to navigate the repair. ChameleonEC finally re-tunes the repair plans to timely bypass emerging stragglers. Extensive experiments show that ChameleonEC can accelerate data repair and shorten access latency of foreground requests.

VIII. Acknowledgments

This work was supported in part by the National Key R&D Program of China (2022YFB4501200), Major Research Plan of the National Natural Science Foundation of China (No. 92373114), Natural Science Foundation of China (No. 62072381), and Natural Science Foundation of Fujian Province of China (No. 2023J06001).

References

[1] "HBase," https://hbase.apache.org/.
[2] "Intel(R) Intelligent Storage Acceleration Library," https://github.com/intel/isa-l.
[3] "Jerasure: Erasure Coding Library," https://jerasure.org/.
[4] "Redis," https://redis.io/.
[5] "The Network is the New Storage Bottleneck," https://www.datanami.com/2016/11/10/network-new-storage-bottleneck/.
[6] "Backblaze Open-sources Reed-Solomon Erasure Coding Source Code," https://www.backblaze.com/blog/reed-solomon/, 2015.
[7] "Yahoo Cloud Object Store - Object Storage at Exabyte Scale," https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at, 2015.
[8] "Erasure Code Support," https://docs.openstack.org/swift/latest/overview_erasure_code.html, 2019.
[9] "Apache Hadoop 3.1.4," https://hadoop.apache.org/docs/r3.1.4/, 2020.
[10] "IBM Object Store Trace," http://iotta.snia.org/traces/key-value/36305, 2020.
[11] "Memcached," http://iotta.snia.org/traces/key-value/28652, 2020.
[12] "Amazon EC2," https://aws.amazon.com/ec2/, 2021.
[13] "HDFS Erasure Coding," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2021.
[14] "Nethogs," https://github.com/raboof/nethogs, 2022.
[15] "The Wonder Shaper 1.4.1," https://github.com/magnific0/wondershaper, 2022.
[16] "Declustered RAID," https://www.ibm.com/docs/en/storage-scale-ece/5.1.9?topic=features-declustered-raid/, 2023.
[17] "OpenStack," https://www.openstack.org/, 2024.
[18] B. Atikoglu, Y. Xu, E. Frachtenberg, J. Song, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012.
[19] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron, "Pelican: A Building Block for Exascale Cold Data Storage," in *Proc. of USENIX OSDI*, 2014.
[20] X. Cao, G. Yang, Y. Gu, C. Wu, J. Li, G. Xue, M. Guo, Y. Dong, and Y. Zhao, "EC-Scheduler: A Load-Balanced Scheduler to Accelerate the Straggler Recovery for Erasure Coded Storage Systems," in *Proc. of ACM/IEEE IWQoS*, 2021.
[21] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *Proc. of USENIX FAST*, 2020.
[22] Z. Cheng, S. Han, P. P. Lee, X. Li, J. Liu, and Z. Li, "An In-Depth Correlative Study between DRAM Errors and Server Failures in Production Data Centers," in *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, 2022.
[23] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer, "Tiered replication: A cost-effective alternative to full cluster geo-replication," in *Prof. of USENIX ATC*, 2015.
[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of ACM SoCC*, 2010.
[25] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.
[26] D. Du, B. Yang, Y. Xia, and H. Chen, "Accelerating extra dimensional page walks for confidential computing," in *Prof. of MICRO*, 2023.
[27] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's Time To Revisit LRU vs. FIFO," in *Proc. of USENIX HotStorage*, 2020.
[28] J. Fang, S. Wan, and X. He, "RAFI: Risk-Aware Failure Identification to Improve the RAS in Erasure-coded Data Centers," in *Prof. of USENIX ATC*, 2018.
[29] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the Locality of Codeword Symbols," *IEEE Transactions on Information theory*, vol. 58, no. 11, pp. 6925–6934, 2012.
[30] S. Han, P. P. Lee, F. Xu, Y. Liu, C. He, and J. Liu, "An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers," in *Proc. of USENIX FAST*, 2021.
[31] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds." in *Proc. of USENIX FAST*, 2012.
[32] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen, "Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage," in *Proc. of USENIX FAST*, 2021.
[33] Y. Hu, P. P. C. Lee, K. W. Shum, and P. Zhou, "Proxy-assisted Regenerating Codes with Uncoded Repair for Distributed Storage Systems," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2512–2528, 2017.
[34] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng, "Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice," *ACM Transactions on Storage*, vol. 13, no. 4, p. 33, 2017.
[35] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *Proc. of USENIX ATC*, 2012.
[36] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie, "PUSH: A Pipelined Reconstruction I/O for Erasure-Coded Storage Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 516–526, 2014.
[37] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *Proc. of USENIX FAST*, 2019.
[38] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads." in *Proc. of USENIX FAST*, 2012.
[39] J. Kim, K. Lim, Y. Jung, S. Lee, C. Min, and S. H. Noh, "Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays." in *Proc. of USENIX ATC*, 2019.
[40] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg, "On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes," in *Proc. of USENIX ATC*, 2018.
[41] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby, "HashTag Erasure Codes: From Theory to Practice," *IEEE Transactions on Big Data*, vol. 4, no. 4, pp. 516–529, 2017.
[42] Y. Lee, H. Al Maruf, M. Chowdhury, A. Cidon, and K. G. Shin, "Hydra: Resilient and Highly Available Remote Memory," in *Proc. of USENIX FAST*, 2022.
[43] G. Lefebvre and M. J. Feeley, "Separating Durability and Availability in Self-Managed Storage," in *Proc. of ACM SIGOPS European Workshop*, 2004.
[44] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu, "More than capacity: Performance-oriented evolution of pangu in alibaba," in *Proc. of USENIX FAST*, 2023.
[45] R. Li, X. Li, P. P. C. Lee, and Q. Huang, "Repair Pipelining for Erasure-Coded Storage," in *Proc. of USENIX ATC*, 2017.
[46] X. Li, K. Cheng, K. Tang, P. P. Lee, Y. Hu, D. Feng, J. Li, and T.-Y. Wu, "ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage," in *Proc. of USENIX FAST*, 2023.
[47] X. Li, R. Li, P. P. C. Lee, and Y. Hu, "OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems," in *Proc. of USENIX FAST*, 2019.
[48] S. Lin, G. Gong, Z. Shen, P. P. C. Lee, and J. Shu, "Boosting Full-Node Repair in Erasure-Coded Storage," in *Proc. of USENIX ATC*, 2021.
[49] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*. Elsevier, 1977, vol. 16.
[50] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage," in *Proc. of ACM EuroSys*, 2016.
[51] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebook's Warm BLOB Storage System," in *Proc. of USENIX OSDI*, 2014.
[52] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast Crash Recovery in RAMCloud," in *Proc. of ACM SOSP*, 2011.

[53] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[54] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. Gad, and Z. Bandic, "Opening the Chrysalis: On the Real Repair Performance of MSR Codes," in *Proc. of USENIX FAST*, 2016.

[55] J. S. Plank and C. Huang, "Tutorial: Erasure Coding for Storage Applications," in *Proc. of USENIX FAST*, 2013.

[56] J. S. Plank, E. L. Miller, K. M. Greenan, B. A. Arnold, J. A. Burnum, A. W. Disney, and A. C. McBride, "GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic," *Tech. Rep. CS-13-716, University of Tennessee*, 2013.

[57] M. Qin, A. N. Reddy, P. V. Gratz, R. Pitchumani, and Y. S. Ki, "KVRAID: High Performance, Write Efficient, Update Friendly Erasure Coding Scheme for KV-SSDs," in *Proc. of ACM SYSTOR*, 2021.

[58] K. V. Rashmi, N. Shah, and P. V. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at The MSR and MBR Points via A Product-Matrix Construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[59] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *Proc. of USENIX OSDI*, 2016.

[60] K. Rashmi, P. Nakkiran, J. Wang, N. Shah, and K. Ramchandran, "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth." in *Proc. of USENIX FAST*, 2015.

[61] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-Coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *Proc. of USENIX HotStorage*, 2013.

[62] ——, "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers," in *Proc. of ACM SIGCOMM*, 2014.

[63] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[64] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring Elephants: Novel Erasure Codes for Big Data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[65] Y. Shan, K. Chen, T. Gong, L. Zhou, T. Zhou, and Y. Wu, "Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair," in *Proc. of ACM SOSP*, 2021, pp. 457–471.

[66] Z. Shen and P. P. C. Lee, "Cross-Rack-Aware Updates in Erasure-Coded Data Centers," in *Proc. of ACM ICPP*, 2018.

[67] Z. Shen, J. Shu, Z. Huang, and Y. Fu, "ClusterSR: Cluster-Aware Scattered Repair in Erasure-Coded Storage," in *Proc. of IEEE IPDPS*, 2020.

[68] Z. Shen, J. Shu, and P. P. C. Lee, "Reconsidering Single Failure Recovery in Clustered File Systems," in *Proc. of IEEE/IFIP DSN*, 2016.

[69] H. Shi, X. Lu, D. Shankar, and D. K. Panda, "UMR-EC: A Unified and Multi-Rail Erasure Coding Library for High-Performance Distributed Storage Systems," in *Proc. of IEEE HPDC*, 2019.

[70] K. W. Shum and Y. Hu, "Functional-Repair-by-Transfer Regenerating Codes," in *Proc. of IEEE ISIT*, 2012.

[71] I. Tamo and A. Barg, "A Family of Optimal Locally Recoverable Codes," *IEEE Transactions on Information Theory*, vol. 60, no. 8, pp. 4661–4676, 2014.

[72] I. Tamo, Z. Wang, and J. Bruck, "Zigzag Codes: MDS Array Codes with Optimal Rebuilding," *IEEE Transactions on Information Theory*, vol. 59, no. 3, pp. 1597–1616, 2012.

[73] C. Tian, B. Sasidharan, V. Aggarwal, V. A. Vaishampayan, and P. V. Kumar, "Layered Exact-Repair Regenerating Codes via Embedded Error Correction and Block Dsesigns," *IEEE Transactions on Information Theory*, vol. 61, no. 4, pp. 1933–1947, 2015.

[74] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy *et al.*, "Clay Codes: Moulding MDS Codes to Yield an MSR Code," in *Proc. of USENIX FAST*, 2018.

[75] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "Infinicache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proc. of USENIX FAST*, 2020.

[76] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu, "Dayu: Fast and Low-Interference Data Recovery in Very-Large Storage Systems," in *Proc. of USENIX ATC*, 2019.

[77] Z. Wang, J. Sim, E. Lim, and J. Zhao, "Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems," in *Prof. of HPCA*, 2022.

[78] H. Weatherspoon and J. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *Proc. of IPTPS*, 2002.

[79] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proc. of USENIX OSDI*, 2006.

[80] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *Proc. of USENIX FAST*, 2017.

[81] J. Yang, Y. Yue, and K. V. Rashmi, "A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter," in *Proc. of USENIX OSDI*, 2020.

[82] H. Zhang, M. Dong, and H. Chen, "Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication," in *Proc. of USENIX FAST*, 2016.

[83] Y. Zhou, H. M. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy *et al.*, "Carbink: Fault-Tolerant Far Memory," in *Proc. of USENIX OSDI*, 2022.

A. Artifact Appendix

A.1 Abstract

We present ChameleonEC, a general mechanism that can assist a variety of erasure codes in realizing low-interference repair. ChameleonEC comprises the following design techniques: (i) repair task assignment, which decomposes a repair plan into multiple repair tasks and makes them co-exist harmoniously with the foreground traffic to saturate unoccupied bandwidth and avoid bandwidth contentions; (ii) repair path establishment, which orchestrates elastic transmission routings over the dispatched repair tasks to instruct the repair; and (iii) straggler-aware re-scheduling, which timely re-tunes task transmissions and repair plans to bypass unexpected stragglers emerging in data repair. We conduct extensive experiments on Amazon EC2, showing that ChameleonEC can accelerate the repair by 4.9–498.2% for various erasure codes under different real-world traces. ChameleonEC can also speed up the repair process by 25.4–73.5% under the storage-bottlenecked scenarios.

A.2 Artifact check-list (meta-information)

- **Program: Jerasure Library v2.0, GF-complete, Redis**
- **Compilation: GCC, python3**
- **Run-time environment: Ubuntu 16.04.7 LTS**
- **Data set: YCSB-A on HBase, IBM Object Store Trace, Memcached, Facebook's ETC workloads on Memcached**
- **Hardware: four vCPU with 3.1 GHz Intel Xeon Platinum processor, 16 GB RAM, and 300 GB General Purpose SSD volumes**
- **Metrics: repair throughput, P99 latency**
- **Publicly available?: The source code of ChameleonEC can be reached via https://github.com/shenzr/ChameleonEC**

A.3 Description

A.3.1 How to access

The source code of ChameleonEC can be reached via https://github.com/shenzr/ChameleonEC. We have prepared a ready-made test environment on Alibaba Cloud.

A.3.2 Hardware dependencies

ChameleonEC need 20 m5.xlarge instances. Each instance runs Ubuntu 16.04.7 LTS and owns four vCPU with 3.1 GHz Intel Xeon Platinum processor, 16 GB RAM, and 300 GB General Purpose SSD volumes. The storage bandwidth is around 500 MB/s and the network bandwidth is 10 Gb/s.

A.3.3 Software dependencies

Jerasure Library v2.0 and GF-complete are used for encoding and decoding. Redis is used for data storage across nodes.

A.3.4 Data sets

YCSB-A on HBase, IBM Object Store Trace, Memcached, Facebook's ETC workloads on Memcached.

A.4 Installation

You can set up Jerasure Library v2.0, GF-complete, and Redis by downloading the codes from their Github and website. You need to install Hadoop and HBase for YCSB-A, and Memcached for their traces.

A.5 Experiment workflow

To perform Hadoop and Hbase:

1. start-all.sh
2. start-hbase.sh

To perform Memcached:

1. service Memcached stop
2. memcached -d install -m 8192 -u root -p 11211 -c 4096 -I 1M

To perform ChemeleonEC:

1. python3 run-BD-Monitor.py
2. python3 run.py 0 (0 for trace-only, 1 for repair-only, 2 for both trace and repair)

A.6 Evaluation and expected results

Most of our results are reported as repair throughput and P99 latency, which are outputs from executing scripts on the host. Summary of major results as follows.

- ChameleonEC improves the repair throughput by 4.9–498.2% and reduces P99 latency by 8.4% (Exp#1–#13);
- ChameleonEC reduces the increase degree of trace execution times by 6.2–81.4% (Exp#2) and gains higher repair throughput under a smaller T_{phase} (Exp#3);
- ChameleonEC can promptly respond to trace changes and gain a high repair throughput (Exp#4);
- ChameleonEC generates repair plans quickly (Exp#5);
- ChameleonEC gains a higher repair throughput than the repair algorithms using RepairBoost (Exp#6);
- ChameleonEC still improves the repair throughput by 25.0–41.3% with no foreground traffic (Exp#7);
- ChameleonEC retains its effectiveness for multi-node repair (Exp#8) and various erasure codes (Exp#9);
- ChameleonEC can accelerate repair for degraded reads (Exp#10) and storage-bottlenecked scenarios (Exp#12);
- The design techniques in ChameleonEC are orthogonal and complementary to each other (Exp#11).